C++/Python library for population genetics.

This library offers *two* simulation packages for population genetics: one for low-dimensional simulations (up to ~15 loci) and one for high-dimensional ones.

Each package is based on a big class that represents a population:

- `haploid_lowd` for low-dimensional populations
- `haploid_highd` for high-dimensional simulations

A simple example routine is the following:

```python
# EXAMPLE SCRIPT
#
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim as h

c = h.haploid_lowd(4)
c.set_allele_frequencies([0, 0.3, 0.6, 0.9], N=1000)
c.evolve(100)
c.plot_diversity_histogram()
plt.show()
```

which evolves a population with 4 loci for 100 generations starting from fixed allele frequencies, under neutral conditions, and plots the diversity histogram afterwards.

For more usage examples, please consult the `tests` and `examples` folders.
1.1 Requirements

FFpopSim is developed to work on 32 or 64 bit machines running Linux or Mac OSX. The basic library is written in C++ and can be used and extended independently of the Python bindings.

The Python bindings are also distributed as ready-to-use binary files (see Install); you can still build the library yourself though, if you prefer to do so.

- On Linux, FFPopSim is expected to be compatible with all distributions, provided they are up to date (glibc 2.14 recommended).
- On Mac OSX, only Intel CPUs are expected to work, and only Mac OSX 10.6 or later. If you have an earlier Mac computer, you can still try to build the library yourself (see Install), but that might fail.

1.1.1 Runtime Requirements

- Python 2.7 (no Python 3 support yet): older Python versions will never be supported, but Python 3 might become so in the future. If you have only Python 2.6 or earlier, consider using the EPD Python distribution or updating your system.
- NumPy 1.6: if your Python distribution has only NumPy 1.5 or earlier, consider using the EPD Python distribution, building FFPopSim from source, or updating your system. It is recommended to import numpy explcitely before using the library, as shown in the examples.
- matplotlib is used in the plot functions. As long as you do not call those functions, you can live without it. However, it is recommended to import it explcitely before using the library, as shown in the examples.

Note: The Enthought Python Distribution (EPD) is a widely used and well-maintained Python environment that provides all necessary Python packages for running FFPopSim, including a recent NumPy version (but no GSL and BOOST). A basic EPD version is available for free at the following website:


1.1.2 Building Requirements

In order to build the Python bindings to FFPopSim, you need the following programs:

- a C++ compiler, e.g. GCC
• Python 2.7+ (no Python 3 support yet), including header files
• NumPy, including header files and shared libraries
• GSL, the GNU Scientific library
• BOOST, the C++ extension library
• an implementation of Make, e.g. GNU Make
• distutils, a library for installing Python packages

In addition, if you modify the sources and want to regenerate the Python bindings, you will need the following programs:
• SWIG, the Simplified Wrapper and Interface Generator

Finally, if you want to rebuild the documentation, you will need the following programs:
• Sphinx, the Python documentation generator, for Python 2.x

The building process has been tested on Python 2.7, Numpy 1.6, gcc 4.7, gsl 1.15, boost 1.50. The regeneration part has been tested on SWIG 2.0. The documentation has been created with Sphinx 1.1.

1.2 Install

**Warning:** If you have not browsed the Requirements section already, please do so, to make sure you have installed all the requirements for FFPopSim.

The installation of the FFPopSim Python module is done via the programs Make and distutils. Please refer also to the INSTALL file if the instructions below do not satisfy your needs (or generate errors!).

1.2.1 Using the binaries

The simplest way to install FFPopSim is using the binaries provided for 32 and 64 bit Linux and Mac OSX (10.6+) systems. You can choose either way:

1. copy manually the files from build/<your arch> into a folder included in your PYTHONPATH, where <your arch> is your architecture, i.e. Linux or Mac and 32 or 64 bit, or
2. if you have distutils, install it system-wide, calling the following command as a superuser:

   ```
   make python-install
   ```

   Neither of these strategies will involve any building, hence you do not need GSL nor BOOST to install the binaries.

   If you want to install FFPopSim using distutils, but into a different location than the standard third-party Python packages directory, you can call directly (if needed, as superuser):

   ```
   python2.7 setup.py install --skip-build --install-lib=<target_dir>
   ```

   where <target_dir> is the target installation directory.

1.2.2 Building FFPopSim locally

To build the module locally, call in your shell:
**1.2.3 Testing FFPopSim**

To test whether FFPopSim is installed correctly (and inserted into your PYTHONPATH), you can open Python2.7 in a new shell and call:

```python
import FFPopSim
```

If you do not get any errors, the installation is successful. You can proceed to the First steps with FFPopSim section.

**1.2.4 Troubleshooting**

In case of problems with the installation, please check in the Requirements section that you have all necessary run-time packages.

Please consult the file INSTALL in the main package folder for further help.

**1.3 First steps with FFPopSim**

FFPopSim is supposed to be easy to use. This page is meant to help new users to familiarize themselves with the library by means of examples. For a complete reference of classes and functions, please see the Contents page.

**1.3.1 For the impatient ones...**

An effective way to discover all available methods is to import FFPopSim from the iPython interactive shell, create a population, and use TAB autocompletion:

```python
In [1]: import FFPopSim as h
In [2]: pop = h.haploid_lowd(5)  # create a population with 5 loci
In [3]: pop. <--- TAB
```

**1.3.2 Importing FFPopSim**

FFPopSim is a single Python module. As such, you can import it with the python import statement, provided the files FFPopSim.py and _FFPopSim.so are in a folder in your PYTHONPATH. If you wish to perform a system-wide installation of FFPopSim, call the make recipe python-install as a superuser:

```bash
$ sudo make python-install
```

**Note:** if this sounds new to you, just put those two files into your current directory, from which you plan to call the Python interpreter. import statements first look in the current folder for modules.

We recommend to import Numpy and matplotlib together with FFPopSim. In short, all your scripts should begin with the following piece of code:
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim

The selective import of parts of FFPopSim using `from FFPopSim import <xxx>` is discouraged and its results are untested.

### 1.3.3 Examples

See the main page for *examples*.

### 1.4 Contents

The functionality of *FFPopSim* is described in detail in the following sections.

#### 1.4.1 Population Class Overviews

Simulations in *FFPopSim* are centered around population classes. These methods of these classes are divided in categories and described at the following pages:

**haploid_lowd Overview**

The following lists all methods and attributes of *haploid_lowd*, grouped according to purpose and function. Member functions are shown with the prefix "haploid_lowd.*". If you have initialized for example as `pop=haploid_lowd(L)`, the prefix in your program needs to be `pop.*`.

**Note:** Clicking on the name of the function will take you to a more detailed explanation listing all arguments.

**Initialization**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_lowd.<strong>init</strong></td>
<td>Construct a low-dimensional population with certain parameters.</td>
</tr>
<tr>
<td>haploid_lowd.copy</td>
<td>Copy population into new instance.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_lowd.L</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.number_of_loci</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.N</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.population_size</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.generation</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.carrying_capacity</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.circular</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.outcrossing_rate</td>
<td></td>
</tr>
<tr>
<td>haploid_lowd.recombination_model</td>
<td>Model of recombination to use</td>
</tr>
</tbody>
</table>
Status

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.status</code></td>
<td>Print a status list of the population parameters</td>
</tr>
</tbody>
</table>

Initialize the Population

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.set_wildtype</code></td>
<td>Initialize population of N individuals with the - allele at all loci (wildtype)</td>
</tr>
<tr>
<td><code>haploid_lowd.set_allele_frequencies</code></td>
<td>Initialize the population in linkage equilibrium with specified allele frequencies.</td>
</tr>
<tr>
<td><code>haploid_lowd.set_genotypes</code></td>
<td>Initialize population with fixed counts for specific genotypes.</td>
</tr>
</tbody>
</table>

Set the fitness landscape

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.set_fitness_additive</code></td>
<td>Set the fitness landscape for individual genotypes.</td>
</tr>
<tr>
<td><code>haploid_lowd.set_fitness_function</code></td>
<td>Set the fitness landscape for individual genotypes.</td>
</tr>
<tr>
<td><code>haploid_lowd.set_fitness_coefficients</code></td>
<td>Set the fitness landscape in Fourier space for individual Fourier coefficients.</td>
</tr>
</tbody>
</table>

Mutation and Recombination

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.circular</code></td>
<td></td>
</tr>
<tr>
<td><code>haploid_lowd.outcrossing_rate</code></td>
<td></td>
</tr>
<tr>
<td><code>haploid_lowd.recombination_model</code></td>
<td>Model of recombination to use</td>
</tr>
<tr>
<td><code>haploid_lowd.set_recombination_rates</code></td>
<td>Set the recombination rate(s).</td>
</tr>
<tr>
<td><code>haploid_lowd.get_recombination_rates</code></td>
<td>Get recombination rates.</td>
</tr>
<tr>
<td><code>haploid_lowd.set_mutation_rates</code></td>
<td>Set the mutation rate(s).</td>
</tr>
<tr>
<td><code>haploid_lowd.get_mutation_rates</code></td>
<td>Get one or several mutation rates.</td>
</tr>
</tbody>
</table>

Evolution

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.evolve</code></td>
<td>Evolve for some generations</td>
</tr>
<tr>
<td><code>haploid_lowd.evolve_deterministic</code></td>
<td>Evolve for some generations deterministically (skips the resampling)</td>
</tr>
<tr>
<td><code>haploid_lowd.evolve_norec</code></td>
<td>Evolve for some generations without recombination</td>
</tr>
</tbody>
</table>

Random Sampling

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.random_genomes</code></td>
<td>Get random genomes according sampled from the population.</td>
</tr>
</tbody>
</table>

Get allele/genotype frequencies

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>haploid_lowd.get_allele_frequency</code></td>
<td>Get the frequency of the + allele</td>
</tr>
<tr>
<td><code>haploid_lowd.get_allele_frequencies</code></td>
<td>Get the frequencies of all + alleles</td>
</tr>
<tr>
<td><code>haploid_lowd.get_genotype_frequency</code></td>
<td>Get the frequency of a genotype</td>
</tr>
<tr>
<td><code>haploid_lowd.get_genotype_frequencies</code></td>
<td>Get the frequency of each genotype.</td>
</tr>
</tbody>
</table>
Table 1.9 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_lwd.get_pair_frequency</td>
<td>Get the frequency of genotypes with the + allele at both loci.</td>
</tr>
<tr>
<td>haploid_lwd.get_ch1</td>
<td>Get chi of an allele in the +/- basis.</td>
</tr>
<tr>
<td>haploid_lwd.get_ch12</td>
<td>Get $\chi_{ij}$</td>
</tr>
<tr>
<td>haploid_lwd.get_LD</td>
<td>Get linkage disequilibrium</td>
</tr>
<tr>
<td>haploid_lwd.get момент</td>
<td>Get moment of two alleles in the +/- basis</td>
</tr>
<tr>
<td>haploid_lwd.allele_entropy</td>
<td></td>
</tr>
<tr>
<td>haploid_lwd.genotype_entropy</td>
<td></td>
</tr>
</tbody>
</table>

Analyze the fitness distribution

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_lwd.get_fitness</td>
<td>Get fitness values of a genotype</td>
</tr>
<tr>
<td>haploid_lwd.get_fitnesses</td>
<td>Get the fitness of all possible genotypes.</td>
</tr>
<tr>
<td>haploid_lwd.get_fitness_statistics</td>
<td></td>
</tr>
<tr>
<td>haploid_lwd.get_fitness_histogram</td>
<td>Get the histogram of the fitness of a sample from the population.</td>
</tr>
<tr>
<td>haploid_lwd.plot_fitness_histogram</td>
<td>Plot the histogram of the fitness of a sample from the population.</td>
</tr>
</tbody>
</table>

Divergence and Diversity

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_lwd.get_divergence_statistics</td>
<td>Get the mean and variance of the divergence of a population sample – same</td>
</tr>
<tr>
<td>haploid_lwd.get_diversity_statistics</td>
<td>Get the mean and variance of the diversity of a population sample.</td>
</tr>
<tr>
<td>haploid_lwd.get_divergence_histogram</td>
<td>Get the histogram of the divergence of a population sample.</td>
</tr>
<tr>
<td>haploid_lwd.get_diversity_histogram</td>
<td>Get the histogram of the diversity in a sample from the population.</td>
</tr>
<tr>
<td>haploid_lwd.plot_divergence_histogram</td>
<td>Plot the histogram of the divergence of a population sample.</td>
</tr>
<tr>
<td>haploid_lwd.plot_diversity_histogram</td>
<td>Plot the histogram of the diversity of a population sample.</td>
</tr>
</tbody>
</table>

haploid_highd Overview

The following lists all methods and attributes of haploid_highd, grouped according to purpose and function. Member functions are shown with the prefix haploid_highd.*. If you have initialized for example as pop=haploid_highd(L), the prefix in your program needs to be pop.*.

Note: Clicking on the name of the function will take you to a more detailed explanation listing all arguments.

Initialization, copy, and storage

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_highd.<strong>init</strong></td>
<td>Construct a high-dimensional population with certain parameters.</td>
</tr>
<tr>
<td>haploid_highd.copy</td>
<td>Copy population into new instance.</td>
</tr>
<tr>
<td>haploid_highd.dump</td>
<td>Dump a population to binary file, for later use.</td>
</tr>
<tr>
<td>load_haploid_highd</td>
<td>Load a population from a compressed pickle file</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haploid_highd.L</td>
<td></td>
</tr>
</tbody>
</table>
Table 1.13 – continued from previous page

| haploid_highd.number_of_loci | haploid_highd.N | haploid_highd.population_size | haploid_highd.number_of_traits | haploid_highd.generation | haploid_highd.participation_ratio | haploid_highd.number_of_clones | haploid_highd.carrying_capacity | haploid_highd.circular | haploid_highd.outcrossing_rate | haploid_highd.crossover_rate | haploid_highd.recombination_model | haploid_highd.mutation_rate | mutation rate (per site per generation) | haploid_highd.trait_weights | weight of each trait on fitness |

Status

| haploid_highd.status | Print a status list of the population parameters |

Initialize the population

| haploid_highd.set_wildtype | Initialize a population of wildtype individuals |
| haploid_highd.set_allele_frequencies | Initialize the population according to the given allele frequencies in linkage equilibrium. |
| haploid_highd.set_genotypes | Initialize population with fixed counts for specific genotypes. |
| haploid_highd.add_genotype | Add new individuals to the population with certain genotypes |

Set phenotypes and fitness function

| haploid_highd.trait_weights | weight of each trait on fitness |
| haploid_highd.set_trait_additive | Set the additive part of a trait |
| haploid_highd.add_trait_coefficient | |
| haploid_highd.set_random_trait_epistasis | |
| haploid_highd.clear_trait | |
| haploid_highd.clear_traits | |

The following methods are shortcuts if you have only one trait and will not work for populations with more than one trait:

| haploid_highd.set_fitness_additive | Shortcut for set_trait_additive when there is only one trait |
| haploid_highd.add_fitness_coefficient | |
| haploid_highd.set_random_epistasis | |
| haploid_highd.clear_fitness | |

Evolution

1.4. Contents
haploid_highd.evolve  Evolve for some generations.
haploid_highd.bottleneck  Make the population undergo a bottleneck
haploid_highd.unique_clones
haploid_highd.flip_single_locus
haploid_highd.calc_stat

Random Sampling

haploid_highd.random_clone  Get a random clone
haploid_highd.random_clones  Get random clones
haploid_highd.random_genomes  Get a sample of random genomes from the population

Clone structure and genotypes

haploid_highd.get_clone_size  Get the size of a clone
haploid_highd.get_clone_sizes  Get the size of all clones.
haploid_highd.get_genotype  Get all genotypes of the population.

Get allele frequencies and linkage disequilibria

haploid_highd.get_allele_frequency  Get all allele frequencies
haploid_highd.get_allele_frequencies
haploid_highd.get_pair_frequency
haploid_highd.get_chi
haploid_highd.get_LD
haploid_highd.get_chi2
haploid_highd.get_moment

Analyze fitness and trait distributions

haploid_highd.calc_stat
haploid_highd.get_trait_additive  Get an array with the additive coefficients of all loci of a trait.
haploid_highd.get_trait_additive
haploid_highd.get_trait_epistasis
haploid_highd.get_trait_statistics
haploid_highd.get_trait_covariance
haploid_highd.get_fitness  Get the fitness of an individual
haploid_highd.get_fitnesses  Get the fitness of all clones.
haploid_highd.get_fitness_statistics
haploid_highd.get_fitness_histogram  Calculate the fitness histogram of a population sample.
haploid_highd.plot_fitness_histogram  Plot a distribution of fitness of a population sample.

Divergence and Diversity
haploid_highd.get_divergence_statistics
haploid_highd.get_diversity_statistics
haploid_highd.get_divergence_histogram
haploid_highd.get_diversity_histogram
haploid_highd.plot_divergence_histogram
haploid_highd.plot_diversity_histogram

Genealogies and Trees

haploid_highd.track_locus_genealogy
haploid_highd.genealogy

Genealogy of the tracked loci.

hivpopulation Overview

hivpopulation is a subclass of haploid_highd, hence inherits its methods. Moreover, hivpopulation has some HIV-specific methods.

hivpopulation has two phenotypic traits, replication and resistance. They contribute linearly to the viral fitness. The relative weight is set by the attribute treatment.

Note: Clicking on the name of the function will take you to a more detailed explanation listing all arguments.

Initialization

hivpopulation.__init__ Construct a HIV population with certain parameters.

Drug Treatment

hivpopulation.treatment

Replication and Resistance

hivpopulation.set_trait_landscape Set HIV trait landscape according to some general parameters.

hivpopulation.set_replication_landscape Set the phenotypic landscape for the replication capacity of HIV.

hivpopulation.set_resistance_landscape Set the phenotypic landscape for the drug resistance of HIV.

hivpopulation.get_replication_additive The additive part of the replication landscape.

hivpopulation.get_resistance_additive The additive part of the resistance landscape.

hivpopulation.set_replication_additive Set the additive replication coefficients

hivpopulation.set_resistance_additive Set the additive drug resistance coefficients

I/O Convenience Functions
1.4.2 Genealogies

FFPopSim can genealogies at individual loci within haploid_highd, using multi_locus_genealogy. Internally, this class itself contains instances of rooted_tree.

The documentation for the genealogy tracking capability is divided in two parts:

- see the website of our review for examples and scripts;
- see haploid_highd or the reference pages below for details on specific functions.

1.4.3 Reference Pages

The classes used in FFPopSim are listed in the following pages for reference purposes:

class clone (n_traits=0)
Clone of isogenetic individuals

trait
Traits vector of the clone

class genotype_value_pair (genotype=[], value=0)
Pair of a genotype and a value

class haploid_lowd (L=1, rng_seed=0)
Class for low-dimensional population genetics (short genomes ~20 loci).

The class offers a number of functions, but an example will explain the basic idea:

```
# EXAMPLE SCRIPT
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim as h

c = h.haploid_lowd(5)  # 5 loci

c.set_genotypes([[0, 2], [300, 700]])
```
# set an additive fitness landscape with these coefficients
```
c.set_fitness_additive([0.02, 0.03, 0.04, 0.02, -0.03])
```
# Note: we are in the -/+ basis, so
# \( F[10000] - F[00000] = 2 \times 0.02 \)
# Hence the coefficients are half of the effect of mutation on fitness
```
c.evolve(100)  # evolve for 100 generations
c.plot_diversity_histogram()
plt.show()
```

---

__init__(L=1, rng_seed=0)
Construct a low-dimensional population with certain parameters.

Parameters:
- L : number of loci (at least 1)
- rng_seed : seed for the random number generator

__str__()  

__repr__()  


copy (rng_seed=0)
Copy population into new instance.

Parameters:
- rng_seed: random number to initialize the new population

status ()
Print a status list of the population parameters

L
number_of_loci

N
population_size
geneneration
circular
carrying_capacity

recombination_model
Model of recombination to use

Available values:
- FFPopSim.FREE_RECOMBINATION: free shuffling between parents
- FFPopSim.CROSSOVERS: block recombination with crossover probability
- FFPopSim.SINGLE_CROSSOVER: block recombination with crossover probability

outcrossing_rate

set_allele_frequencies (frequencies, N)
Initialize the population in linkage equilibrium with specified allele frequencies.

Parameters:
- frequencies: an array of length L with all allele frequencies
- N: set the population size and, if still unset, the carrying capacity to this value

**Note:** the population size is only used for resampling and has therefore no effect on the speed of the simulation.

**set_genotypes**(*genotypes*, *counts*)

Initialize population with fixed counts for specific genotypes.

**Parameters:**

- *genotypes*: list of genotypes to set. Genotypes are specified as integers, from 00...0 that is 0, up to 11...1 that is \(2^{L}-1\).
- *counts*: list of counts for those genotypes

**Note:** the population size and, if unset, the carrying capacity will be set as the sum of the counts.

**Note:** you can use Python binary notation for the indices, e.g. 0b0110 is 6.

**set_wildtype**(*N*)

Initialize population of *N* individuals with the - allele at all loci (wildtype)

**Parameters:**

- *N*: the number of individuals

**Note:** the carrying capacity is set to the same value if still unset.

**set_recombination_rates**(*rates*, *model=None*)

Set the recombination rate(s).

**Parameters:**

- *rates*: if a double, the recombination rate at between any two loci; if an array, the locus-specific recombination rates
- *model*: the recombination model to use (CROSSOVERS or, for linear genomes, SINGLE_CROSSOVER)

**Note:** if locus-specific rates are specified, the array must have length \(L-1\) for linear chromosomes and length \(L\) for circular ones. The \(i\)-th element is the crossover rate between the \(i\)-th site and the \((i+1)\)-th site.

**Note:** if the recombination model is not specified, the current model will be kept or, if the current model is FREE_RECOMBINATION, then CROSSOVERS will be set.

**get_recombination_rates**()

Get recombination rates.

**Returns:**

- the rates between neighboring loci, a list of float of length \(L-1\)
**set_mutation_rates** (*rates, rates_back=None*)
Set the mutation rate(s).

**Parameters:**
- **rates:** if a double, the mutation rate at any locus in both directions or, if rates_back is not None, only in the forward direction
  - if a vector, the mutation rate is specified for each locus, the same in both directions or, if rates_back is not None, only in the forward direction
- **rates_back:** mutation rate in the backward direction (global or locus-specific)

**get_mutation_rates** (*locus=None, direction=None*)
Get one or several mutation rates.

**Parameters:**
- **locus:** get only the mutation rate(s) of this locus
- **direction:** get only the forward or backward mutation rate(s). This argument is a Boolean, 0/False for forward rates, 1/True for backward rates.

**Returns:**
- the mutation rate(s) requested

**Note:** if the mutation rates for all loci and/or directions are the same, this function will try to be smart and give you the answer you are looking for. In case of doubt, you will get a matrix (L x 2) with the full mutation rate landscape.

**evolve** (*gen=1*)
Evolve for some generations

**Parameters:**
- **gen:** number of generations to evolve the population, defaults to one

**evolve_deterministic** (*gen=1*)
Evolve for some generations deterministically (skips the resampling)

**Parameters:**
- **gen:** number of generations to evolve the population

**evolve_norec** (*gen=1*)
Evolve for some generations without recombination

**Parameters:**
- **gen:** number of generations to evolve the population

**get_genotype_frequency** (*genotype*)
Get the frequency of a genotype

**Parameters:**
- **genotype:** genotype, whose the frequency is to be returned

**Returns:**
- the frequency of the genotype
**get_genotype_frequencies()**
Get the frequency of each genotype.

**get_allele_frequency(\textit{locus})**
Get the frequency of the + allele

**Parameters:**
- \textit{locus}: locus, at which the frequency of the + allele is to be computed

**Returns:**
- the frequency of the + allele, $u_i := \frac{1 + s_i}{2}$, where $s_i \in \{-1, 1\}$.

**get_allele_frequencies()**
Get the frequencies of all + alleles

**get_pair_frequency(\textit{locus1}, \textit{locus2})**
Get the frequency of genotypes with the + allele at both loci.

**Parameters:**
- \textit{locus1}: first locus
- \textit{locus2}: second locus

**Returns:**
- the joint frequency of the + alleles

**get_chi(\textit{locus})**
Get chi of an allele in the -/+ basis

**Parameters:**
- \textit{locus}: locus whose chi is to be computed

**Returns:**
- the chi of that allele, $\chi_i := s_i$, where $s_i \in \{-1, 1\}$.

**get_chi2(\textit{locus1}, \textit{locus2})**
Get $\chi_{ij}$

**Parameters:**
- \textit{locus1}: first locus
- \textit{locus2}: second locus

**Returns:**
- the linkage disequilibrium between them, i.e. $\chi_{ij} := s_i s_j - s_i \cdot s_j$.

**get_LD(\textit{locus1}, \textit{locus2})**
Get linkage disequilibrium

**Parameters:**
- \textit{locus1}: first locus
• locus2: second locus

**Returns:**
• the linkage disequilibrium between them, i.e. \( D_{ij} := \frac{1}{4} \left< s_i s_j \right> - \chi_i \cdot \chi_j \).  

**get_moment** (locus1, locus2)
Get moment of two alleles in the -/+ basis

**Parameters:**
• locus1: first locus
• locus2: second locus

**Returns:**
• the second moment, i.e. \( \left< s_i s_j \right> \), where \( s_i, s_j \in \{-1, 1\} \).

**random_genomes** (n_sample)
Get random genomes according sampled from the population.

**Parameters:**
• n_sample: number of random genomes to sample

**Returns:**
• integers corresponding to random genomes in the population.

**get_fitness** (genotype)
Get fitness values of a genotype

**Parameters:**
• genotype: genotype whose fitness is to be calculated. This can either be an integer or in binary format, e.g. 5 = 0b101

**Returns:**
• the fitness of that genotype.

**get_fitnesses** ()
Get the fitness of all possible genotypes.

**get_fitness_histogram** (n_sample=1000, **kwargs)
Get the histogram of the fitness of a sample from the population.

**Parameters:**
• n_sample: number of individual to sample at random from the population. defaults to 1000

**Returns:**
• h: numpy.histogram of fitness in the population

**plot_fitness_histogram** (axis=None, n_sample=1000, **kwargs)
Plot the histogram of the fitness of a sample from the population.

**Parameters:**
• axis: use an already existing axis for the plot
• n_sample: number of individual to sample at random from the population. Defaults to 1000.
• kwargs: further optional keyword arguments to \texttt{numpy.histograms}

\textbf{get\_divergence\_statistics} \texttt{(n\_sample=1000)}

Get the mean and variance of the divergence of a population sample – same as mean and variance of allele frequencies.

\textbf{Parameters:}

• \texttt{n\_sample}: number of individuals to sample at random from the population. defaults to 1000.

\textbf{Returns:}

• \texttt{stat}: structure with mean and variance of divergence in the population

\textbf{get\_divergence\_histogram} \texttt{(bins=10, n\_sample=1000, **kwargs)}

Get the histogram of the divergence of a population sample.

\textbf{Parameters:}

• \texttt{bins}: number of bins or list of bin edges (passed verbatim to \texttt{numpy.histogram})

• \texttt{n\_sample}: number of individual to sample at random from the population, defaults to 1000.

• \texttt{kwargs}: further optional keyword arguments to \texttt{numpy.histograms}

\textbf{Returns:}

• \texttt{h}: \texttt{numpy.histogram} of divergence in the population

\textit{Note:} to get a normalized histogram, use the \texttt{density} keyword.

\textbf{plot\_divergence\_histogram} \texttt{(axis=None, n\_sample=1000, **kwargs)}

Plot the histogram of the divergence of a population sample.

\textbf{Parameters:}

• \texttt{axis}: use an already existing axis for the plot

• \texttt{n\_sample}: number of individual to sample at random from the population, defaults to 1000.

• \texttt{kwargs}: further optional keyword arguments to \texttt{numpy.histograms}

\textbf{get\_diversity\_statistics} \texttt{(n\_sample=1000)}

Get the mean and variance of the diversity of a population sample

\textbf{Parameters:}

• \texttt{n\_sample}: number of individual to sample at random from the population, defaults to 1000.

\textbf{Returns:}

• \texttt{stat}: structure with mean and variance of diversity in the population

\textbf{get\_diversity\_histogram} \texttt{(bins=10, n\_sample=1000, **kwargs)}

Get the histogram of the diversity in a sample from the population.

\textbf{Parameters:}

• \texttt{bins}: number of bins or list of bin edges (passed verbatim to \texttt{numpy.histogram})

• \texttt{n\_sample}: number of individual to sample at random from the population, defaults to 1000.

• \texttt{kwargs}: further optional keyword arguments to \texttt{numpy.histograms}

\textbf{Returns:}

• \texttt{h}: \texttt{numpy.histogram} of diversity in the population

\textit{Note:} to get a normalized histogram, use the \texttt{density} keyword.
**plot_diversity_histogram** *(axis=None, n_sample=1000, **kwargs)*
Plot the histogram of the diversity of a population sample.

**Parameters:**
- axis: use an already existing axis for the plot
- n_sample: number of individual to sample at random from the population, defaults to 1000.
- kwargs: further optional keyword arguments to numpy.histograms

**set_fitness_function** *(genotypes, values)*
Set the fitness landscape for individual genotypes.

**Parameters:**
- genotypes: genotype to which the fitness values will be assigned. Genotypes are specified as integers, from 00...0 that is 0, up to 11...1 that is 2^L-1.
- values: fitness values to assign

**Note:** you can use Python binary notation for the genotypes, e.g. 0b0110 is 6.

**set_fitness_coefficients** *(coefficients, values)*
Set the fitness landscape in Fourier space for individual Fourier coefficients.

**Parameters:**
- coefficients: Fourier coefficients to which the values will be assigned. They are specified as integers, from 00...0 that is 0, up to 11...1 that is 2^L-1.
- values: values to assign

**Note:** you can use Python binary notation for the coefficients, e.g. 0b0110 is 6.

**set_fitness_additive** *(coefficients)*

**genotype_entropy** ()

**allele_entropy** ()

**haploid_highd Class Reference**

**haploid_highd** offers a number of methods, listed below. Global functions related to **haploid_highd** are shown after the methods.

**Contents**
- haploid_highd Class Reference
  - Class methods
  - Global functions related to haploid_highd

**Class methods**

**class haploid_highd** *(L=0, rng_seed=0, number_of_traits=1, all_polymorphic=False)*
Class for high-dimensional population genetics (genomes larger than ~20 loci).
This class is the main object for simulating the evolution of populations with many loci (more than ~20). The class offers a number of functions, but an example will explain the basic idea:

```python
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim as h

c = h.haploid_highd(300)  # 300 loci
pop.set_wildtype(1000)    # start with 1000 wildtype individuals
pop.mutation_rate = 1e-4  # mutation rate per site per generation
pop.outcrossing_rate = 1e-1 # probability of sexual reproduction per gen
pop.crossover_rate = 1e-2  # probability of crossover per site per gen
pop.evolve(100)           # evolve for 100 generations

c.plot_divergence_histogram()
plt.show()
```

Populations can have a number of phenotypic traits that contribute to the fitness of each individual. The function that calculates fitness from the phenotype identifies fitness with the first trait only by default. The user is, however, free to subclass haploid_highd in C++ (as it is done in hivpopulation) and implement their own phenotype -> fitness function.

In addition, the trait landscapes describe the genotype -> phenotype maps. These can be set directly from Python (since the genotypic space has a finite number of elements).

**Note:** fitness is not a phenotypic trait directly, but rather a function of all phenotypic traits together.

---

__init__(L=0, rng_seed=0, number_of_traits=1, all_polymorphic=False)

Construct a high-dimensional population with certain parameters.

**Parameters:**

- L: number of loci
- rng_seed: seed for the random generator. If zero (default) pick a random number
- number_of_traits: number of phenotypic traits, defaults to one
- all_polymorphic: option to use an infinite-sites model tracking ancestral alleles (only available with a single phenotypic trait and zero mutation rate)

__str__()
__repr__()

copy (rng_seed=0)

Copy population into new instance.

**Parameters:**

- rng_seed: random number to initialize the new population

dump (filename, format='bz2', include_genealogy=False)

Dump a population to binary file, for later use.

**Parameters:**

- filename: the path to the file where to store the information
- format: one of ‘bz2’ or ‘plain’. Choose the former if you want compression.
- include_genealogy: if True, the multi_locus_genealogy is stored as well (if present).
Note: The population can be reloaded using the function FFPopSim.load_haploid_highd.

```status()
```
Print a status list of the population parameters

```L
number_of_loci
N
population_size
generation
number_of_clones
number_of_traits
max_fitness
participation_ratio
circular
carrying_capacity
recombination_model
outcrossing_rate
crossover_rate
mutation_rate
mutation rate (per site per generation)
```
```
trait_weights
weight of each trait on fitness
```

Note: Fitness is updated automatically when the weights are changed.

```get_clone(n)
```

```set_wildtype(N)
```
Initialize a population of wildtype individuals

Parameters:

- N: the number of individuals

Note: the carrying capacity is set to the same value if still unset.

```set_allele_frequencies(frequencies, N)
```
Initialize the population according to the given allele frequencies in linkage equilibrium.

Parameters:

- frequencies: an array of length L with all allele frequencies
- N: set the population size and, if still unset, the carrying capacity to this value
**FFPopSim Documentation, Release 2.0**

**set_genotypes** *(genotypes, counts)*

Initialize population with fixed counts for specific genotypes.

**Parameters:**

- genotypes: list of genotypes to set. Genotypes are lists of alleles, e.g. [[0,0,1,0], [0,1,1,1]] for genotypes 0010 and 0111
- counts: list of the number at which each of those genotypes it to be present

**Note:** the population size and, if unset, the carrying capacity will be set as the sum of the counts.

**Example:** if you want to initialize **200 individuals with genotype 001** and **300 individuals with genotype 110**, you can use `set_genotypes([[0,0,1], [1,1,0]], [200, 300])`

**evolve** *(gen=1)*

Evolve for some generations.

**Parameters:**

- gen: number of generations, defaults to one

**bottleneck** *(size_of_bottleneck)*

Make the population undergo a bottleneck

**Parameters:**

- size_of_bottleneck: the number of individuals at the bottleneck

**flip_single_locus** *(locus)*

**calc_stat** *

**get_divergence_statistics** *(n_sample)*

**get_diversity_statistics** *(n_sample)*

**get_trait_statistics** *(t)*

**get_fitness_statistics** *

**get_trait_covariance** *(t1, t2)*

**get_allele_frequency** *(locus)*

**get_allele_frequencies** *

Get all allele frequencies

**get_pair_frequency** *(locus1, locus2)*

**get_chisq** *(locus)*

**get_chisq2** *(locus1, locus2)*

**get_LD** *(locus1, locus2)*

**get_moment** *(locus1, locus2)*

**add_genotype** *(genotype, n=1)*

Add new individuals to the population with certain genotypes

**Parameters:**

- genotype: genotype to add to the population (Boolean list)
• n: number of new individuals carrying that genotype

**get\_trait\_additive** \((t)\)
Get an array with the additive coefficients of all loci of a trait.

**Parameters:**
• t: number of the trait

**Returns:**
• coefficients: array of additive coefficients for the selected trait

**clear\_trait** \((t)\)
**clear\_traits** ()
**clear\_fitness** ()

**set\_trait\_additive** \((coefficients, t)\)
Set the additive part of a trait

**Parameters:**
• coefficients: array of coefficients for the trait (of length L). All previous additive coefficients are erased
• t: number of the trait to set

**set\_fitness\_additive** \((coefficients)\)
Shortcut for set\_trait\_additive when there is only one trait

**add\_trait\_coefficient** \((value, loci, t=0)\)
**add\_fitness\_coefficient** \((value, loci)\)
**get\_trait\_epistasis** \((t=0)\)
**set\_random\_trait\_epistasis** \((epistasis\_std, t=0)\)
**set\_random\_epistasis** \((epistasis\_std)\)

**get\_fitness** \((n)\)
Get the fitness of an individual

**Parameters:**
• n: index of the clone whose fitness is to be computed

**Returns:**
• fitness: fitness value of that clone

**get\_fitnesses** ()
Get the fitness of all clones.

**get\_trait** \((n, t=0)\)
Get a trait of an individual

**Parameters:**
• n: index of the clone whose trait is to be computed
• t: trait to be computed

**Returns:**
• trait: value of that trait for that clone
get_traits()
Get all traits from all clones

get_clone_size\((n)\)
Get the size of a clone

Parameters:
• n: index of the clone

Returns:
• size: size of the selected clone

gget_clone_sizes()
Get the size of all clones.

get_genotype\((n)\)
get_genotype(haploid_highd self, int n) -> boost::dynamic_bitset<>

get_genotypes()
Get all genotypes of the population.

Return:
• genotypes: boolean 2D array with the genotypes

Note: this function does not return the sizes of each clone.

unique_clones()

distance_Hamming\((clone_gt1, clone_gt2, chunks=None, every=1)\)
Calculate the Hamming distance between two genotypes

Parameters:
• clone_gt1: index of the clone corresponding to the first genotype
• clone_gt2: index of the clone corresponding to the second genotype
• chunks: list of pairs delimiting the genetic areas to include
• every: do the comparison only on certain sites

Example: to calculate the distance between the first two clones limited to third codon positions between locus 90 and 200, use: distance_Hamming(0, 1, chunks=[92, 200], every=3).

random_genomes\((n)\)
Get a sample of random genomes from the population

Parameters:
• n: number of random genomes to compute

Returns:
• gts: (n x L) bool matrix with the n genotypes

random_clone()
Get a random clone

Returns:
• clone: index of the random clone
**random clones**\((n)\)
Get random clones

**Parameters:**
- \(n\): number of random clones to return

**Returns:**
- clones: clone indices

**get_fitness_histogram**\((n_sample=1000, **kwargs)\)
Calculate the fitness histogram of a population sample.

**Parameters:**
- n_sample: number of individuals to sample

**Returns:**
- h: numpy.histogram of fitness in the population

**plot_fitness_histogram**\((axis=None, n_sample=1000, **kwargs)\)
Plot a distribution of fitness of a population sample.

**Parameters:**
- axis: an axis to use. A new figure is created by default
- n_sample: number of individuals to sample
- kwargs: further optional keyword arguments to matplotlib.pyplot.hist

**Returns:**
- return value of axis.hist(...)

**get_divergence_histogram**\((bins=10, chunks=None, every=1, n_sample=1000, **kwargs)\)
Get the divergence histogram restricted to those chunks of the genome.

**Parameters:**
- bins: number or array of bins to be used in the histogram (see also numpy.histogram)
- chunks: **restrict analysis to some chunk in the genome. It must be an \(n \times 2\) matrix with** the initial and (final+1) positions of the chunks
- every: **restrict analysis to every \(X\) positions. For instance, if every third site is neutral, this argument can be used to only look at those neutral sites**
- n_sample: number of individuals to sample
- kwargs: further optional keyword arguments to numpy.histogram

**Returns:**
- h: numpy.histogram of divergence in the population

**plot_divergence_histogram**\((axis=None, n_sample=1000, **kwargs)\)
Plot the divergence histogram of a population sample.

**Parameters:**
- axis: an axis to use. A new figure is created by default
- n_sample: number of individuals to sample
- kwargs: further optional keyword arguments to matplotlib.pyplot.hist
Returns:

- return value of axis.hist(...)

**get_diversity_histogram** *(bins=10, chunks=None, every=1, n_sample=1000, **kwargs)*

Get the diversity histogram restricted to those chunks of the genome.

Parameters:

- **bins**: number or array of bins to be used in the histogram (see also numpy.histogram)
- **chunks**: restrict analysis to some chunk in the genome. It must be an n x 2 matrix with the initial and (final+1) positions of the chunks
- **every**: restrict analysis to every X positions. For instance, if every third site is neutral, this argument can be used to only look at those neutral sites
- **n_sample**: number of individuals to sample
- **kwargs**: further optional keyword arguments to numpy.histogram

Returns:

- h: numpy.histogram of diversity in the population

**plot_diversity_histogram** *(axis=None, n_sample=1000, **kwargs)*

Plot the diversity histogram of a population sample.

Parameters:

- **axis**: an axis to use. A new figure is created by default
- **n_sample**: number of individuals to sample
- **kwargs**: further optional keyword arguments to matplotlib.pyplot.hist

Returns:

- return value of axis.hist(...)

**genealogy**

Genealogy of the tracked loci.

**Note**: This attribute is read-only.

**track_locus_genealogy** *(loci)*

Global functions related to haploid_highd

**load_haploid_highd** *(filename, gen_loci=[], include_genealogy=False)*

Load a population from a compressed pickle file

Parameters:

- **filename**: the path of the pickle file
- **gen_loci**: start tracking these loci in the population
- **include_genealogy**: load the old genealogy if present
**hivgene Class reference**

```python
class hivgene (start_in=0, end_in=10000, second_start_in=0, second_end_in=0)

Structure for an HIV gene.
```

**hivpopulation Class Reference**

```python
class hivpopulation (N=0, rng_seed=0, mutation_rate=3e-05, coinfection_rate=0.01,
crossover_rate=0.001)

Class for HIV population genetics (genome size = 10000).
```

This class is the main object for simulating the evolution of HIV. The class offers a number of functions, but an example will explain the basic idea:

```python
# EXAMPLE SCRIPT
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim as h

c = h.hivpopulation(2000)  # Create a population of 2000 individuals
c.evolve(100)  # Evolve (neutrally) for 100 generations
c.plot_divergence_histogram()
plt.show()
```

This class is a subclass of haploid_high and offers most of its methods. In addition to the haploid_high class, this class offers functions for reading fitness and drug resistance landscapes from a text file, and to save genomes as plain text or in compressed NumPy format.

Moreover, there are two phenotypic traits, replication and resistance. Their relative importance for viral fitness is set by the `treatment` attribute:

```python
f[trait] = trait[0] + treatment * trait[1]
```

By default, `treatment` is set to zero, to simulate non-treated patients.

The gene structure of HIV is modelled roughly, including only start/end positions for the exons, using HXB2 as a reference. Different genes do not get automatically different fitness landscapes.

```python
__init__ (N=0, rng_seed=0, mutation_rate=3e-5, coinfection_rate=1e-2, crossover_rate=1e-3)

Construct a HIV population with certain parameters.
```

Parameters:
- `N` number of viral particles
- `rng_seed` seed for the random number generator. If this is 0, time(NULL)+getpid() is used.
- `mutation_rate` mutation rate in events / generation / site
- `coinfection_rate` probability of coinfection of the same cell by two viral particles in events / generation
- `crossover_rate` probability of template switching during coinfection in events / site

**Note:** the genome length is 10000 (see HIVGENOME).

```python
__str__ ()
```
__repr__()

treatment

write_genotypes (filename, sample_size, gt_label='', start=0, length=0)

write_genotypes_compressed (filename, sample_size, gt_label='', start=0, length=0)

Store random genotypes into a compressed file.

Parameters:

• filename: string with the name of the file to store the genotype into
• sample_size: how many random genotypes to store
• gt_label: common fasta label for the genotypes (e.g. “HIV-sim”)
• start: if only a portion of the genome is to be stored, start from this position
• length: store a chunk from start to this length

The genotypes can be read using numpy.load.

read_replication_coefficients (filename)

read_resistance_coefficients (filename)

set_trait_landscape (traitnumber=0, lethal_fraction=0.05, deleterious_fraction=0.8, adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1, effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0, epitope_strength=0.05, number_valleys=0, valley_strength=0.1)

Set HIV trait landscape according to some general parameters.

Parameters:

• lethal_fraction: fraction of lethal sites
• deleterious_fraction: fraction of deleterious sites
• adaptive_fraction: fraction of beneficial sites
• effect_size_lethal: effect of lethal changes
• effect_size_deleterious: average effect of deleterious changes
• effect_size_adaptive: average effect of beneficial changes
• env_fraction: fraction of beneficial sites in env
• effect_size_env: average effect of beneficial changes in env
• number_epitopes: number of (epistatic) epitopes
• epitope_strength: average height of an epitope escape mutation
• number_valleys: number of (epistatic) valleys
• valley_strength: average depth of a valley

Note: the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.
**Note:** fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

**Note:** the third positions are always neutral (synonymous).

```python
set_replication_landscape(lethal_fraction=0.05, deleterious_fraction=0.8,
adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1,
effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0,
epitope_strength=0.05, number_valleys=0, valley_strength=0.1)
```

Set the phenotypic landscape for the replication capacity of HIV.

**Parameters:**
- lethal_fraction: fraction of lethal sites
- deleterious_fraction: fraction of deleterious sites
- adaptive_fraction: fraction of beneficial sites
- effect_size_lethal: effect of lethal changes
- effect_size_deleterious: average effect of deleterious changes
- effect_size_adaptive: average effect of beneficial changes
- env_fraction: fraction of beneficial sites in env
- effect_size_env: average effect of beneficial changes in env
- number_epitopes: number of (epistatic) epitopes
- epitope_strength: average height of an epitope escape mutation
- number_valleys: number of (epistatic) valleys
- valley_strength: average depth of a valley

**Note:** the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.

**Note:** fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

**Note:** the third positions are always neutral (synonymous).

```python
set_resistance_landscape(lethal_fraction=0.05, deleterious_fraction=0.8,
adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1,
effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0,
epitope_strength=0.05, number_valleys=0, valley_strength=0.1)
```

Set the phenotypic landscape for the drug resistance of HIV.

**Parameters:**
• lethal_fraction: fraction of lethal sites
• deleterious_fraction: fraction of deleterious sites
• adaptive_fraction: fraction of beneficial sites
• effect_size_lethal: effect of lethal changes
• effect_size_deleterious: average effect of deleterious changes
• effect_size_adaptive: average effect of beneficial changes
• env_fraction: fraction of beneficial sites in env
• effect_size_env: average effect of beneficial changes in env
• number_epitopes: number of (epistatic) epitopes
• epitope_strength: average height of an epitope escape mutation
• number_valleys: number of (epistatic) valleys
• valley_strength: average depth of a valley

Note: the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.

Note: fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

Note: the third positions are always neutral (synonymous).

get_replication_additive()
The additive part of the replication landscape.

Returns:
• coefficients: array of additive replication coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

get_resistance_additive()
The additive part of the resistance landscape.

Returns:
• coefficients: array of additive drug resistance coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

set_replication_additive(coefficients)
Set the additive replication coefficients

Parameters:
• coefficients: array of additive replication coefficients
Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

`set_resistance_additive(coefficients)`
Set the additive drug resistance coefficients

Parameters:
- coefficients: array of additive drug resistance coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

`copy(rng_seed=0)`
Copy population into new instance.

Parameters:
- rng_seed: random number to initialize the new population

`get_replication_additive()`
The additive part of the replication landscape.

Returns:
- coefficients: array of additive replication coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

`get_resistance_additive()`
The additive part of the resistance landscape.

Returns:
- coefficients: array of additive drug resistance coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

`set_replication_additive(coefficients)`
Set the additive replication coefficients

Parameters:
- coefficients: array of additive replication coefficients

Warning: the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

`set_replication_landscape(lethal_fraction=0.05, deleterious_fraction=0.8, adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1, effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0, epitope_strength=0.05, number_valleys=0, valley_strength=0.1)`
Set the phenotypic landscape for the replication capacity of HIV.

Parameters:
• lethal_fraction: fraction of lethal sites
• deleterious_fraction: fraction of deleterious sites
• adaptive_fraction: fraction of beneficial sites
• effect_size_lethal: effect of lethal changes
• effect_size_deleterious: average effect of deleterious changes
• effect_size_adaptive: average effect of beneficial changes
• env_fraction: fraction of beneficial sites in env
• effect_size_env: average effect of beneficial changes in env
• number_epitopes: number of (epistatic) epitopes
• epitope_strength: average height of an epitope escape mutation
• number_valleys: number of (epistatic) valleys
• valley_strength: average depth of a valley

**Note:** the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.

**Note:** fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

**Note:** the third positions are always neutral (synonymous).

**set_resistance_additive**(coefficients)
Set the additive drug resistance coefficients

Parameters:

• coefficients: array of additive drug resistance coefficients

**Warning:** the -/+ basis is used throughout the library. If you are used to the 0/1 basis, keep in mind that the interaction series-expansion is different.

**set_resistance_landscape**(lethal_fraction=0.05, deleterious_fraction=0.8, adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1, effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0, epitope_strength=0.05, number_valleys=0, valley_strength=0.1)
Set the phenotypic landscape for the drug resistance of HIV.

Parameters:

• lethal_fraction: fraction of lethal sites
• deleterious_fraction: fraction of deleterious sites
• adaptive_fraction: fraction of beneficial sites
• effect_size_lethal: effect of lethal changes
• effect_size_deleterious: average effect of deleterious changes
• effect_size_adaptive: average effect of beneficial changes
• env_fraction: fraction of beneficial sites in env
• effect_size_env: average effect of beneficial changes in env
• number_epitopes: number of (epistatic) epitopes
• epitope_strength: average height of an epitope escape mutation
• number_valleys: number of (epistatic) valleys
• valley_strength: average depth of a valley

**Note:** the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.

**Note:** fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

**Note:** the third positions are always neutral (synonymous).

**set_trait_landscape** (traitnumber=0, lethal_fraction=0.05, deleterious_fraction=0.8, adaptive_fraction=0.01, effect_size_lethal=0.8, effect_size_deleterious=0.1, effect_size_adaptive=0.01, env_fraction=0.1, effect_size_env=0.01, number_epitopes=0, epitope_strength=0.05, number_valleys=0, valley_strength=0.1)

Set HIV trait landscape according to some general parameters.

**Parameters:**

• lethal_fraction: fraction of lethal sites
• deleterious_fraction: fraction of deleterious sites
• adaptive_fraction: fraction of beneficial sites
• effect_size_lethal: effect of lethal changes
• effect_size_deleterious: average effect of deleterious changes
• effect_size_adaptive: average effect of beneficial changes
• env_fraction: fraction of beneficial sites in env
• effect_size_env: average effect of beneficial changes in env
• number_epitopes: number of (epistatic) epitopes
• epitope_strength: average height of an epitope escape mutation
• number_valleys: number of (epistatic) valleys
• valley_strength: average depth of a valley

**Note:** the effects of deleterious and beneficial sites are exponentially distributed, i.e. most of them will still be almost neutral.
Note: fractions refer to first and second positions only. For instance, by default, 80% of first and second positions outside env are deleterious.

Note: the third positions are always neutral (synonymous).

write_genotypes_compressed (filename, sample_size, gt_label='', start=0, length=0)

Store random genotypes into a compressed file.

Parameters:

- filename: string with the name of the file to store the genotype into
- sample_size: how many random genotypes to store
- gt_label: common fasta label for the genotypes (e.g. “HIV-sim”)
- start: if only a portion of the genome is to be stored, start from this position
- length: store a chunk from start to this length

The genotypes can be read using numpy.load.

thisown
The membership flag

index_value_pair Class Reference

class index_value_pair (index_in=0, val_in=0)

Pair of an index and a value

multi_locus_genealogy Class Reference

class multi_locus_genealogy
Genealogy for multiple loci

    __init__()  # Default constructor
    __str__()  # __repr__()

    track_locus (new_locus)
    reset()
    reset_but_loci()
    get_tree (locus)

    loci  # The loci that are being tracked
polymorphism Class Reference

class polymorphism (b=0, age=0, e=0, f=0, fvar=0)
Polymorphism history
__str__()
__repr__()
birth
sweep_time
effect
fitness
fitness_variance

rooted_tree Class Reference

class rooted_tree
Rooted phylogenetic tree.
This class is used to represent the phylogenetic tree of a single locus. It is possible to print the tree in Newick
format, to get the subtree spanned by some of the leaves, and to look at the tree nodes and edges.
__str__()
__repr__()

external_branch_length()
total_branch_length()
calc_weight_distribution(subtree_root)
ancestors_at_age(age, subtree)
Find nodes in subtree younger than a certain age
Parameters:
• age: critical age to check
• subtree: subtree to look for nodes in

Returns:
• ancestors: the ancestors at that age

create_subtree_from_keys(leaves)
print_newick()
subtree_newick(subtree_root)
to_Biopython_tree()
Convert the tree into Biopython format
Returns:
• tree: Biopython.Phylo phylogenetic tree representation of self

edges
Edges of the tree
nodes
Nodes of the tree

leafs
Leaves of the tree

stat Class Reference

class stat (mean_in=0, variance_in=0)
Mean and variance of a statistical distribution

tree_key Class Reference

class tree_key (index=0, age=0)
Key for a phylogenetic tree, with index and age.

__init__ (index=0, age=0)
Initialize new tree_key.

Parameters:
• index: index of the key
• age: age of the key

__str__ ()
__repr__ ()

index
age

tree_step Class Reference

class tree_step (pos=0, step=0)
Step in a phylogenetic tree search

__init__ (pos=0, step=0)
Initialize new step.

Parameters:
• pos: position
• step: length of step

__str__ ()
__repr__ ()

pos
step
**tree_node Class Reference**

```python
class tree_node
    Node of a phylogenetic tree
    __str__()
    __repr__()
    parent_node
    own_key
    child_edges
        Child edges of the node
    fitness
    clone_size
    crossover
        Crossover of node
    weight_distribution
        Distribution of weights of this node
```

**tree_edge Class Reference**

```python
class tree_edge
    Edge of a phylogenetic tree
    __str__()
    __repr__()
    parent_node
    own_key
    segment
        Segment of edge
    length
    number_of_offspring
```

In addition, the following helper functions are defined:

**binarify Function Reference**

```python
binarify(gt, L=0)
    Transform an integer into a binary sequence on the L hypercube.
```

**Parameters:**
- gt: integer representing a genotype
- L: number of dimensions of the hypercube

**Returns:**
- genotype: bool vector representing the same genotype

**Examples:**
In [1]: binarify(3, 5)
Out[1]: array([False, False, False, True, True], dtype=bool)

In [2]: FFPopSim.binarify(0b11, 5)
Out[2]: array([False, False, False, True, True], dtype=bool)

integerify Function Reference

**integerify**($b$)

Transform a binary sequence on the HC into an integer.

**Parameters:**

- $b$: bool vector representing a genotype

**Returns:**

- $gt$: integer representing the same genotype

**Examples:**

In [1]: integerify([False, True, True])
Out[1]: 3

In addition, the underlying C++ library is documented here. Note that some objects and functions have slightly different names in C++ and Python.
Usage examples of FFPopSim can be found at the following pages. The descriptions focus on FFPopSim and tend to ignore aesthetic aspects of the scripts such as figures, labels, et similia. This also means that gluing together the various code chunks found at those pages will not produce exactly the same figures; neither is this necessary at all, because the full source code for all examples (and more) can be found in the examples folder.

Note: examples are ordered by increasing complexity.

2.1 Low-dimensional examples

2.1.1 Decay of linkage disequilibrium

In recombining populations, genetic linkage decays with time. This script initializes a population with high linkage disequilibrium (LD) and tracks how LD. The example can be found in decay_of_LD.py.

First, we load the FFPopSim module, along with the number crunching and plotting tools:

```python
import numpy as np
import matplotlib.pyplot as plt
import FFPopSim as h
```

Next, we set up the population:

```python
# specify parameters
N = 500000  # Population size
L = 4       # number of loci
mu = 0.0    # no new mutations
r = 0.01    # recombination rate

### set up
pop = h.haploid_lowd(L)  # produce an instance of haploid_lowd with L loci
pop.carrying_capacity = N # set the steady-state population size
pop.set_recombination_rates(r)  # assign the recombination rates
pop.set_mutation_rates(mu)  # assign the mutation rate

# initialize the population with N/2 individuals with genotypes 0, that is ---- # and N/2 with the opposite genotype 2**L -1, that is ++++
pop.set_genotypes([0, 2**L-1],[N/2, N/2])
```
Third, we let the population evolve and we track linkage disequilibrium via the `get_LD` function:

```python
max_gen = 50
# get LD for locus pairs (0,1), (0,2) and (0,3)
LD_trajectories = [[pop.generation, pop.get_LD(0,1), pop.get_LD(0,2), pop.get_LD(0,3)]]
for ii in range(max_gen):
    pop.evolve(5)  # evolve 5 generations
    LD_trajectories.append([pop.generation, pop.get_LD(0,1), pop.get_LD(0,2), pop.get_LD(0,3)])
LD_trajectories = np.array(LD_trajectories)  # cast to an array for plotting
```

Fourth, we plot the resulting linkage disequilibrium curves:

```python
cols = ['r', 'b', 'g', 'm', 'c']
for ii in range(LD_trajectories.shape[1]-1):
    # plot the LD from simulations and compare it to the exponential decay expected from theory
    plt.plot(LD_trajectories[:,0], LD_trajectories[:,ii+1], color=cols[ii], label=r'$D_{1'+str(ii+1)+'}$')
    plt.plot(LD_trajectories[:,0], 0.25*np.exp(-LD_trajectories[:,0]* r * (ii+1)), ls='--', color=cols[ii])
plt.legend()
plt.xlabel('Time [generations]')
plt.ylabel('LD $D_{ij}$')
plt.ion()
plt.show()
```

As expected, LD decays faster if loci are further apart. The typical plot we obtain is the following and shows complete concordance of theory and simulations:
2.1.2 Mutation-selection balance

In finite populations, new alleles are introduced continuously by mutation and their abundance subject to genetic drift. If their effects are not neutral, however, selection acts as well. In the steady state, a balance between drift and selection sets in and determines the spectrum of allele frequencies. The full script for this example can be found in the examples folder, in `mutation_selection_balance_lowd.py`.

First, after loading all necessary modules, we set the parameters as usual and create the population class:

```python
N = 500  # population size
L = 4    # number of loci
s = np.linspace(-0.2, 0.7, L) / N  # additive selection coefficients for L loci, scaled to N
mu = 0.4 / N  # mutation rate, scaled to N
r = 5.0 / N  # recombination rate for each interval between loci
pop = h.haploid_lowd(L)  # produce an instance of haploid_lowd with L loci
```

Note that selection coefficients go from negative to positive across the L sites. We set the fitness landscape of the population:

```python
pop.set_fitness_additive(0.5 * s)
```

Note: FFPopSim models fitness landscape in a +/- rather than 0/1 basis, hence the factor 0.5
We set the mutation/recombination rates, using a full multiple-crossover model:

```python
pop.set_mutation_rates(mu)  # mutation rate
pop.set_recombination_rates(r)  # recombination rate (CROSSOVERS model by default)
```

We fix the carrying capacity and initialize the population with wildtypes only:

```python
pop.carrying_capacity = N  # set the population size
pop.set_genotypes([0], [N])  # wildtype individuals, that is ----
```

Now we can start to evolve the population. We first let it equilibrate towards the steady-state:

```python
pop.evolve(10 * N)  # run for 10N generations to equilibrate
```

and we start to record the allele frequencies from now on:

```python
for ii in range(nsamples):
    pop.evolve(0.1 * N)  # N / 10 generations between successive samples
    # get allele frequencies
    allele_frequencies[ii, :] = pop.get_allele_frequencies()
```

Finally, we histogram and plot the spectra of the single sites separately, since they have a different fitness coefficient:

```python
for locus in range(L):
    y, x = np.histogram(allele_frequencies[:, locus], bins=af_bins, density='True')
    plt.plot(bin_centers, y, ...)
```

The result of the plot, compared to diffusion theory, is shown here below.
It fits the simulations quite well indeed; note the accumulation of alleles close to the boundaries. Note also that \( rN \gg 1 \), hence the sites are essentially uncoupled.

### 2.1.3 Time complexity and scaling

Recombination is implemented in `haploid_lowd` in such a way that it scales with the number of loci as \( O(3^L) \) instead of the naive \( O(8^L) \). Moreover, if a single crossover event is allowed to happen, the complexity is reduced even further to \( O(L \cdot 2^L) \). This is shown in this example, called `speed_lowd.py`.

First, modules and paths are imported as usual, plus the `time` module is imported as well:

```python
import time
```

Second, population parameters are set:

```python
N = 1000  # Population size
Lmax = 12  # Maximal number of loci
r = 0.01  # Recombination rate
mu = 0.001 # Mutation rate
G = 1000  # Generations
```

Third, simulations of \( G \) generations are repeated for various number of loci \( L \), to show the scaling behaviour of the recombination algorithm:
exec_time = []
for L in range(2,Lmax):
    t1=time.time()
    ### set up
    pop = h.haploid_lowd(L)  # produce an instance of haploid_lowd with L loci
    pop.carrying_capacity = N  # set the population size
    
    # set and additive fitness function. Note that FFPopSim models fitness landscape
    pop.set_fitness_additive(0.01*np.random.randn(L))
    pop.set_recombination_rates(r)  # assign the recombination rates
    pop.set_mutation_rates(mu)  # assign the mutation rate
    
    #initialize the population with N individuals with genotypes 0, that is ----
    pop.set_allele_frequencies(0.2*np.ones(L), N)
    pop.evolve(G)  # run for G generations
    t2=time.time()

    exec_time.append([L, t2-t1])  # store the execution time

exec_time=np.array(exec_time)

Fourth, the same schedule is repeated with a simpler recombination model, in which only one crossover is allowed, setting the recombination rates with the optional argument pop.set_recombination_rates(r, h.SINGLE_CROSSOVER), and without recombination, using haploid_lowd.evolve_norec instead of the haploid_lowd.evolve.

Fifth, the time required is plotted agains the number of loci and compared to the expectation \(O(3^L)\):

```python
plt.figure()
plt.plot(exec_time[:,0], exec_time[:,1],label='with recombination', linestyle='None', marker = 'o')
plt.plot(exec_time[:,0], exec_time[-1,1]/3.0**(Lmax-exec_time[:,0]-1),label=r'$\propto 3^L$')
plt.plot(exec_time_norec[:,0], exec_time_norec[:,1],label='without recombination', linestyle='None', marker = 'x')
plt.plot(exec_time[:,0], exec_time_norec[-1,1]/2.0**(Lmax-exec_time_norec[:,0]-1),label=r'$\propto 2^L$')
plt.plot(exec_time[:,0], exec_time[-1,1]/3.0**(Lmax)*8**(exec_time[:,0]),label=r'$\propto 8^L$')

ax=plt.gca()
ax.set_yscale('log')
plt.xlabel('number of loci')
plt.ylabel('seconds for '+str(G)+' generations')
plt.legend(loc=2)
plt.xlim([1,Lmax])
plt.ylim([0.2*np.min(exec_time_norec[:,1]),10*np.max(exec_time[:,1])])
plt.ion()
plt.show()
```

The result confirm the theoretical expectation:
2.1.4 Valley Crossing

Now, something slightly more advanced. If sign epistasis is at work in a certain population, the fitness landscape will include valleys. Those valleys need to be crossed by some individual carrying multiple mutations in order to reach the fitness maximum. Recombination can accelerate this process as a source of genetic diversity (in addition to random mutation). This works up to a certain point, where recombination becomes so frequent that it actually destroys beneficial combinations of mutations more often than it creates them. This phenomenon is simulated in this example, which can be found integrally in valley.py.

First, we load the usual modules, and we set the population parameters:

```python
L = 4  # Number of loci
N = 1e10  # Population size
s1 = 1e-5  # Fitness of wildtype
s2 = 0.01  # Fitness of quadruple mutant
```

Then, we decide what recombination and mutation rates to explore:

```python
rs = np.logspace(-4,-3,10).tolist() + \
    [0.00125, 0.0015, 0.00175, 0.002, 0.00225, 0.0025, 0.00275, 0.003, 0.0031, 0.0032, 0.0033, 0.0034, 0.0035, 0.00375, 0.004, 0.005, 0.0075, 0.01]
mutation_rates=[1e-7,1e-6, 1e-5]
```

We then repeat the simulation for various mutation and recombination rates:

```python
for k, mu in enumerate(mutation_rates):
    [...]  
    for i, r in enumerate(rs):
        [...]  
        c = h.haploid_lowd(L)  # produce population with L loci
```

2.1. Low-dimensional examples
c.set_genotypes([0], [N])  # initialize with N individuals in genotype 0 (wildtype)
c.set_recombination_rates(r)  # set the recombination rate
c.set_mutation_rates(mu)

# set the wildtype fitness to s1 and the quadruple mutant fitness to s1+s2 (all other genotypes have relative fitness 0)
c.set_fitness_function([0b0, 0b1111], [s1, s1+s2])

# cross valley: evolve for gens generations at a time until quadruple mutant is above frequency 0.5
gens = 100
while c.get_genotype_frequency(0b1111) < 0.5 and c.generation < 1e6:
    c.evolve(gens)

Finally, we plot all lines with error bars:

ax.errorbar(rs, times, dtimes,
c=colors[k],
lw=2,
label=r'$\mu=10^{'+str(int(np.log10(mu)))+'}$')

We obtain the following plot (note that the calculation takes a while):

![Plot of time for valley crossing](image)

### 2.1.5 Fitness wave

Populations with static, additive fitness landscapes evolve steadily towards the fitness maximum. This example shows this phenomenon using subclassing, which is a central concept in object-oriented programs. The code is located in
As usual, we start by loading the required modules. Before starting the actual script, however, we prepare a new class, \texttt{haploid\_lowd\_track}. This is intended to be a low-dimensional population like \texttt{haploid\_lowd}, but a beefed-up version of it that can also keep track of the fitness histogram over time and plot it. Such an extended variant of an existing class is called a \textit{subclass}.

Defining our extended population amounts to the following pieces of code. First, we define the new class and state clearly that it will steal (or \textit{inherit}) all the features of a normal \texttt{haploid\_lowd} population:

```python
class haploid\_lowd\_track(FFPopSim.haploid\_lowd):
    '''This class tracks the fitness distribution'''
```

Then, we tune the \texttt{\_\_init\_\_} method, which is called when we \textit{initialize} a new instance of a class:

```python
def \_\_init\_\_(self, \*args, \*\*kwargs):
    super(haploid\_lowd\_track, self).\_\_init\_\_(\*args, \*\*kwargs)
    self.fitness\_wave = []
```

We are just saying that we

- initialize the population normally, like the usual class \texttt{haploid\_lowd} (or \textit{superclass}) does, by calling the \texttt{super} line, and
- add an attribute called \texttt{fitness\_wave} that will store the information about the fitness histograms.

In addition, we trick the \texttt{evolve} function a little bit, so that after each call it will store the fitness histogram in our attribute, \texttt{fitness\_wave}:

```python
def evolve(self, \*args, \*\*kwargs):
    ret = super(haploid\_lowd\_track, self).evolve(\*args, \*\*kwargs)
    self.fitness\_wave.append((self.generation,
                                self.get\_fitness\_histogram(density=True,
                                                            n\_sample=50000,
                                                            bins=30)))
    return ret
```

Again, the \texttt{super} line calls the usual \texttt{evolve} function. After evolution has taken place, we store the current fitness histogram in our attribute, and then return control to the user.

**Note:** if you are wondering what all those \texttt{\*args} and \texttt{\*\*kwargs} are, they are only a way to pass all possible argument of that function to the superclass function. It makes the subclass transparent; whatever arguments the original function wanted, they are still valid now, and the subclass does not mess around with them. This is a common technique when you are only modifying superclass functions a little bit.

Finally, we add a handy function that plots the fitness histograms accumulated up to that point:

```python
def plot\_tracks(self):
    '''Plot the tracked histograms'''
    l = len(self.fitness\_wave)
    colors = cm.jet([int(255.0 * i / l) for i in xrange(l)])
    for i, (g, dist) in enumerate(self.fitness\_wave):
        x = dist[1]
        x = 0.5 * (x[:-1] + x[1:])
        y = dist[0] + 1e\-5
        plt.plot(x, y, c=colors[i], lw=2, label='gen '+str(g))
    plt.xlabel('Fitness')
    plt.legend(loc=1)
    plt.yscale('log')
```

2.1. Low-dimensional examples
This whole subclass introduction might look a bit difficult at first, but that’s a normal and healthy reaction if you are not used to object-oriented programming. It is really only a way to keep some properties and functions related to the population in the class itself, instead that out there in the script.

Finally, we can write our script as usual. We define some parameters:

```makefile
N = 500000  # Population size
L = 12      # number of loci
mu = 1e-5   # mutation rate
r = 0.01    # recombination rate
```

we set them in the newly created population (note that we call `haploid_lowd_track` to build the population this time!):

```python
pop = haploid_lowd_track(L)  # produce an instance of haploid_lowd with L loci
pop.carrying_capacity = N     # set the steady-state population size
pop.set_recombination_rates(r, h.SINGLE_CROSSOVER) # assign the recombination rates
pop.set_mutation_rates(mu)    # assign the mutation rate
```

we initialize the individuals and the fitness landscape:

```makefile
# initialize the population with N/2 individuals with genotypes 0, that is ----
# and N/2 with the opposite genotype, that is ++++
pop.set_wildtype(N)

# set a fitness landscape
pop.set_fitness_additive(1e-3 * np.linspace(1, 4, L))
```

and we finally evolve it:

```python
for i in xrange(5):
    pop.evolve(200)
```

Note that we do not have to track the fitness histogram explicitly in the script, because our subclass is doing it internally in its tuned `evolve` method already. Furthermore, we can now exploit the plotting facilities of `haploid_lowd_track`:

```python
# Plot the fitness wave
pop.plot_tracks()
```

and get the following image:
We can see that, as time goes by, the population is able to explore more and more beneficial regions of the fitness landscape, and that the nose of the fitness distribution advances at approximately constant speed, which makes it a travelling wave.

From the algorithmic point of view, we have seen that subclassing is an alternative style than scripting. Instead of keeping track of stuff explicitly in your main script, you do it internally by modifying some class functions or adding new ones altogether.

2.2 High-dimensional examples

2.2.1 Genetic drift

As a very basic example, let's simulate how allele frequencies change due to genetic drift. If we want to track a large number of loci, we can use haploid_hd. First, we import the necessary tools, including FFPopSim:

```python
import numpy as np
from matplotlib import pyplot as plt
import FFPopSim as h
```

Next, we specify the parameters of the population and set up the population.

```python
L=256  # simulate 256 loci
pop = h.haploid_highd(L)  # produce an instance of haploid_lowd with L loci
```
FFPopSim Documentation, Release 2.0

```python
pop.carrying_capacity = 5000  # set the average population size to 5000
pop.outcrossing_rate = 1     # make the species obligate outcrossing
pop.crossover_rate = 0.02/pop.L  # set the crossover rate of the segment to 2 centimorgans
pop.mutation_rate = 0.1/pop.carrying_capacity  # per locus mutation rate equal to 0.1/N

initial_allele_frequencies = 0.5*np.ones(pop.L)  # define some initial allele frequencies

pop.set_allele_frequencies(initial_allele_frequencies, pop.carrying_capacity)

We now have a population of size 5000 where each of the 256 loci is present at frequency 1/2. Next, we want to evolve the population and track the frequencies of the alleles.

```python
maxgen = 2000
allele_frequencies = [pop.get_allele_frequencies()]

while pop.generation<maxgen:
    pop.evolve(10)  # proceed 10 generations
    allele_frequencies.append(pop.get_allele_frequencies())  # save the allele frequencies
    tp.append(pop.generation)  # and the associated generation
```

The array `allele_frequencies` now contains the frequencies of 256 loci every 10 generations. We can plot a subset of these as follows:

```python
for locus in xrange(5,pop.L,50):  # plot a few neutral trajectories
    plt.plot(tp, allele_frequencies[:,locus], c=cm.cool(locus), lw=2)
```

50 Chapter 2. Examples
2.2.2 Genetic drift versus genetic draft

The next examples explores the interplay between genetic drift and genetic draft, i.e., the effect of linked selection on the trajectories of neutral alleles. The basic script is the same as above, only that we now set a fitness landscape and change the effects of some mutations from being deleterious to beneficial during the simulation. This generates selective sweeps.

After importing the relevant modules, we build the population:

```python
L=256  # simulate 256 loci
pop = h.haploid_highd(L)  # produce an instance of haploid_lowd with L loci
pop.carrying_capacity = 50000  # set the average population size to 50000
pop.outcrossing_rate = 1  # make the species obligate outcrossing
pop.crossover_rate = 0.02/pop.L  # set the crossover rate of the segment to 2 centimorgans
pop.mutation_rate = 0.1/pop.carrying_capacity  # per locus mutation rate equal to 0.1/N
```

In addition, we set the selection coefficients to 0 for most loci, but make every 10th locus strongly deleterious:

```python
m=10
selection_coefficients = 0.0*np.ones(pop.L)  # most loci are neutral
selection_coefficients[::m] = -0.1  # every m-th locus is strongly deleterious
pop.set_trait_additive(selection_coefficients,0)  # trait 0 is by default fitness
```
Neutral loci are set the frequency 1/2, while the deleterious ones to frequency 0. We initialize the population with those allele frequencies, in linkage equilibrium:

```python
initial_allele_frequencies = 0.5*np.ones(pop.L)
initial_allele_frequencies[::m] = 0
pop.set_allele_frequencies(initial_allele_frequencies, pop.carrying_capacity)
```

Next, we start evolving and track the allele frequencies as we go along. Every 200 generations, we pick a random locus from the deleterious ones and make it beneficial.

```python
#evolve for 2000 generations and track the allele frequencies
maxgen = 2000
allele_frequencies = [pop.get_allele_frequencies()]
tp = [pop.generation]
while pop.generation<maxgen:
    pop.evolve(10)  # proceed 10 generations
    if (pop.generation%200==0):  # every 200 generations, make one
        print "generation:", pop.generation, 'out of', maxgen
        selection_coefficients[m*np.random.randint(0,25)] = 0.01
        pop.set_trait_additive(selection_coefficients)  # update fitness function
    allele_frequencies.append(pop.get_allele_frequencies())  # save the allele frequencies
    tp.append(pop.generation)  # and the associated generation
```

We now plot the frequency trajectories of all selected loci. Those that become beneficial in the process have risen quickly to high frequencies. When they sweep, they influence the trajectories of linked neutral loci, of which also a few trajectories are shown.

```python
for locus in xrange(0,pop.L,m):  #plot the allele frequency trajectories of the selected mutations
    plt.plot(tp, allele_frequencies[:,locus], c=cm.cool(locus),lw=2)
for locus in xrange(5,pop.L,50):  #plot a few neutral trajectories
    plt.plot(tp, allele_frequencies[:,locus], c=cm.cool(locus), lw=2)
```

52 Chapter 2. Examples
2.2.3 Mutation-selection balance

For sites under selection, both genetic drift and selection influence the fate of new alleles. Similar to the example for `haploid_lowd`, this example shows how to measure the mutation-selection balance in the allele frequency spectra. The full script for this example can be found in the examples folder, in `mutation_selection_balance_highd.py`.

After loading the modules, we start off by setting parameters and constructing the class:

```python
N = 500  # population size
L = 64   # number of loci
s = np.linspace(-2, 2, L) / N  # additive selection coefficients for L loci, scaled to N
mu = 0.5 / N  # mutation rate, scaled to N
r = 50.0 / N / L  # recombination rate for each interval between loci
pop = h.haploid_highd(L)  # produce an instance of haploid_highd with L loci
```

We set the additive fitness landscape. Note that the recombination rate is high enough for loci to be unlinked:

```python
pop.set_fitness_additive(0.5 * s)
```

**Note:** FFPopSim models fitness landscape in a +/- rather than 0/1 basis, hence the factor 0.5
We then set the mutation and recombination rates:

```python
pop.mutation_rate = mu  # mutation rate
pop.recombination_model = h.CROSSOVERS  # recombination model
pop.outcrossing_rate = 1  # obligate sexual
pop.crossover_rate = r  # crossover rate
```

We initialize the population in linkage equilibrium with allele frequencies 0.5:

```python
pop.carrying_capacity = N  # set the population size
pop.set_allele_frequencies(0.5 * np.ones(L), N)
```

Now we can start to evolve the population. We first let it equilibrate towards the steady-state:

```python
pop.evolve(10 * N)  # run for 10N generations to equilibrate
```

and we start to record the allele frequencies from now on:

```python
for ii in range(nsamples):
    pop.evolve(0.1 * N)  # N / 10 generations between successive samples
    allele_frequencies[ii, :] = pop.get_allele_frequencies()
```

Finally, we make a histogram of the allele frequencies and plot it, together with diffusion theory predictions:

```python
for locus in range(L):
    y, x = np.histogram(allele_frequencies[:, locus], bins=af_bins, normed='True')
    plt.plot(bc, y, color=plt.cm.jet(locus*4))
    plt.plot(bc, y/diffusion_theory, color=plt.cm.jet(locus*4), ls='--')
```

The result is shown in the following figures, in the left panel, the allele frequency distribution, in the right panel the same normalized by the diffusion theory prediction:
Comparison to diffusion theory for $rN=0.78125$, $\mu N=0.5$, $N=500$

Color indicates selection coefficient from $Ns=-2..2$
Diffusion theory predicts the spectrum accurately over a wide range of fitness coefficients.

### 2.2.4 Condensation of genotypes driven by epistasis

Epistasis and rugged landscapes favour more clonal populations compared to smooth and monotonic landscapes. This process is observed in this example, which can be found in `condensation.py`.

After importing the modules, we set up the population:

```python
L = 64  # simulate 64 loci
pop = ffpop.haploid_highd(L)  # produce an instance of haploid_lowd with L loci
pop.outcrossing_rate = 0  # make the species asexual
pop.set_random_epistasis(0.05)  # add a random epistatic effect
pop.set_allele_frequencies(np.ones(L)*0.5, 10000)  # initialize the population in LD with allele frequencies 1/2
```

We let the population evolve and collect statistics on fitness, clone size, and participation ratio along the way:

```python
pfit = pop.get_fitness_statistics()
popstat = []
for gen in xrange(1, 200):
    # append current statistics to the list
    pfit = pop.get_fitness_statistics()
    popstat.append([gen, pfit.mean, pfit.variance, pop.participation_ratio, pop.number_of_clones])
    # evolve for dt generations and clean up
```
Finally, we plot some observables on the clone structure of the population, the participation ratio and the number of clones:
Epistasis has the effect of reducing the number of clones over time and, equivalently, increasing the participation ratio. After 200 generations, the number of clones is $O(1)$. 
Indices and tables

- genindex
- modindex
- search
f

FFPopSim, 3
Symbols

__init__() (haploid_highd method), 20
__init__() (haploid_lowd method), 13
__init__() (hivpopulation method), 27
__init__() (multi_locus_genealogy method), 34
__init__() (tree_key method), 36
__init__() (tree_step method), 36
__repr__() (haploid_highd method), 20
__repr__() (haploid_lowd method), 13
__repr__() (hivpopulation method), 27
__repr__() (multi_locus_genealogy method), 34
__repr__() (polymorphism method), 35
__repr__() (rooted_tree method), 35
__repr__() (tree_edge method), 37
__repr__() (tree_key method), 36
__repr__() (tree_node method), 37
__repr__() (tree_step method), 36
__str__() (haploid_highd method), 20
__str__() (haploid_lowd method), 13
__str__() (hivpopulation method), 27
__str__() (multi_locus_genealogy method), 34
__str__() (polymorphism method), 35
__str__() (rooted_tree method), 35
__str__() (tree_edge method), 37
__str__() (tree_key method), 36
__str__() (tree_node method), 37
__str__() (tree_step method), 36

calc_stat() (haploid_highd method), 22
calc_weight_distribution() (rooted_tree method), 35
carrying_capacity (haploid_highd attribute), 21
carrying_capacity (haploid_lowd attribute), 13
child_edges (tree_node attribute), 37
circular (haploid_highd attribute), 21
circular (haploid_lowd attribute), 13
clear_fitness() (haploid_highd method), 23
clear_traits() (haploid_highd method), 23
clear_traits() (haploid_lowd method), 13
cloned (class in FFPopSim), 12
cloned_size (tree_node attribute), 37
copy() (haploid_highd method), 20
copy() (haploid_lowd method), 13
copy() (hivpopulation method), 31
create_subtree_from_keys() (rooted_tree method), 35
crossover (tree_node attribute), 37
crossover_rate (haploid_highd attribute), 21
distance_Hamming() (haploid_highd method), 24
dump() (haploid_highd method), 20
distance() (rooted_tree attribute), 35
effect (polymorphism attribute), 35
evolve() (haploid_highd method), 22
evolve() (haploid_lowd method), 15
evolve_deterministic() (haploid_lowd method), 15
evolve_norec() (haploid_lowd method), 15
external_branch_length() (rooted_tree method), 35
FFPopSim (module), 1
fitness (polymorphism attribute), 35
fitness (tree_node attribute), 37
fitness_variance (polymorphism attribute), 35
flip_single_locus() (haploid_highd method), 22

A
add_fitness_coefficient() (haploid_highd method), 23
add_genotype() (haploid_highd method), 22
add_trait_coefficient() (haploid_highd method), 23
age (tree_key attribute), 36
allele_entropy() (haploid_lowd method), 19
ancestors_at_age() (rooted_tree method), 35

B
binarify() (in module FFPopSim), 37
birth (polymorphism attribute), 35
bottleneck() (haploid_highd method), 22
G
genealogy (haploid_highd attribute), 26
generation (haploid_highd attribute), 21
genotype_entropy() (haploid_lowd attribute), 19
genotype_value_pair (class in FFPopSim), 12
get_allele_frequencies() (haploid_highd method), 22
get_allele_frequencies() (haploid_lowd method), 16
get_allele_frequency() (haploid_highd method), 22
get_allele_frequency() (haploid_lowd method), 16
get_chi() (haploid_highd method), 22
get_chi() (haploid_lowd method), 16
get_chi2() (haploid_highd method), 22
get_chi2() (haploid_lowd method), 16
get_clone() (haploid_highd method), 21
get_clone() (haploid_lowd method), 22
get_clone_size() (haploid_highd method), 24
get_clone_sizes() (haploid_highd method), 24
generate_divergence_histogram() (haploid_highd method), 25
generate_divergence_histogram() (haploid_lowd method), 18
generate_divergence_statistics() (haploid_highd method), 22
generate_divergence_statistics() (haploid_lowd method), 18
generate_diversity_histogram() (haploid_highd method), 26
generate_diversity_histogram() (haploid_lowd method), 18
generate_diversity_statistics() (haploid_highd method), 22
generate_diversity_statistics() (haploid_lowd method), 18
generate_fitness() (haploid_highd method), 23
generate_fitness() (haploid_lowd method), 17
generate_fitness_histogram() (haploid_highd method), 25
generate_fitness_histogram() (haploid_lowd method), 17
generate_fitnesses() (haploid_highd method), 23
generate_fitnesses() (haploid_lowd method), 17
genotype() (haploid_highd method), 24
genotype_frequency() (haploid_lowd method), 15
genotype_frequencies() (haploid_lowd method), 15
genotype() (haploid_highd method), 24
get_LD() (haploid_highd method), 22
get_LD() (haploid_lowd method), 22
genome() (haploid_highd method), 16
genome() (haploid_lowd method), 15
get_moment() (haploid_highd method), 22
get_moment() (haploid_lowd method), 17
get_mutation_rates() (haploid_lowd method), 15
generate_pair_frequency() (haploid_highd method), 15
generate_pair_frequency() (haploid_lowd method), 15
generate_recombination_rates() (haploid_lowd method), 16
generate_reproduction_additive() (hivpopulation method), 30, 31
generate_trait() (haploid_highd method), 30, 31
generate_trait_additive() (hivpopulation method), 30, 31
generate_trait() (haploid_highd method), 23
generate_trait_additive() (haploid_highd method), 23
generate_trait_covariance() (haploid_highd method), 22
generate_trait_epistasis() (haploid_highd method), 23
generate_traits() (haploid_highd method), 22
generate_traits() (haploid_lowd method), 23
get_tree() (multi_locus_genealogy method), 34

H
haploid_highd (class in FFPopSim), 19
haploid_lowd (class in FFPopSim), 12
hivgene (class in FFPopSim), 27
hivpopulation (class in FFPopSim), 27

I
index (tree_key attribute), 36
index_value_pair (class in FFPopSim), 34
integerify() (in module FFPopSim), 38

L
L (haploid_highd attribute), 21
L (haploid_lowd attribute), 13
leaves (rooted_tree attribute), 36
length (tree_edge attribute), 37
load_haploid_highd() (in module FFPopSim), 26
loci (multi_locus_genealogy attribute), 34

M
max_fitness (haploid_highd attribute), 21
multi_locus_genealogy (class in FFPopSim), 34
mutation_rate (haploid_highd attribute), 21

N
N (haploid_highd attribute), 21
N (haploid_lowd attribute), 13
nodes (rooted_tree attribute), 35
number_of_clones (haploid_highd attribute), 21
number_of_loci (haploid_highd attribute), 21
number_of_loci (haploid_lowd attribute), 13
number_of_offspring (tree_edge attribute), 37
number_of_traits (haploid_highd attribute), 21

O
outcrossing_rate (haploid_highd attribute), 21
outcrossing_rate (haploid_lowd attribute), 21
own_key (tree_edge attribute), 37
own_key (tree_node attribute), 37

P
parent_node (tree_edge attribute), 37
parent_node (tree_node attribute), 37
participation_ratio (haploid_highd attribute), 21
plot_divergence_histogram() (haploid_highd method), 25
plot_divergence_histogram() (haploid_lowd method), 18
plot_diversity_histogram() (haploid_highd method), 26
plot_diversity_histogram() (haploid_lowd method), 18
plot_fitness_histogram() (haploid_highd method), 25
plot_fitness_histogram() (haploid_lowd method), 17
polymorphism (class in FFPopSim), 35
population_size (haploid_highd attribute), 21
population_size (haploid_lowd attribute), 13
pos (tree_step attribute), 36
print_newick() (rooted_tree method), 35

random_clone() (haploid_highd method), 24
random_clones() (haploid_highd method), 24
random_genomes() (haploid_highd method), 24
random_genomes() (haploid_lowd method), 17
read_replication_coefficients() (hivpopulation method), 28
read_resistance_coefficients() (hivpopulation method), 28
recombination_model (haploid_highd attribute), 21
recombination_model (haploid_lowd attribute), 13
reset() (multi_locus_genealogy method), 34
reset_but_loci() (multi_locus_genealogy method), 34
rooted_tree (class in FFPopSim), 35

segment (tree_edge attribute), 37
set_allele_frequencies() (haploid_highd method), 21
set_allele_frequencies() (haploid_lowd method), 13
set_fitness_additive() (haploid_highd method), 23
set_fitness_additive() (haploid_lowd method), 19
set_fitness_coefficients() (haploid_lowd method), 19
set_fitness_function() (haploid_lowd method), 19
set_genotypes() (haploid_highd method), 21
set_genotypes() (haploid_lowd method), 14
set_mutation_rates() (haploid_lowd method), 15
set_random_epistasis() (haploid_highd method), 23
set_random_trait_epistasis() (haploid_highd method), 23
set_recombination_rates() (haploid_lowd method), 14
set_replication_additive() (hivpopulation method), 30, 31
set_replication_landscape() (hivpopulation method), 29, 31
set_resistance_additive() (hivpopulation method), 31, 32
set_resistance_landscape() (hivpopulation method), 29, 32
set_trait_additive() (haploid_highd method), 23
set_trait_landscape() (hivpopulation method), 28, 33
set_wildtype() (haploid_highd method), 21
set_wildtype() (haploid_lowd method), 14
stat (class in FFPopSim), 36
status() (haploid_highd method), 21
status() (haploid_lowd method), 13
step (tree_step attribute), 36
subtree_newick() (rooted_tree method), 35
sweep_time (polymorphism attribute), 35

thisown (hivpopulation attribute), 34
to_Biopython_tree() (rooted_tree method), 35
total_branch_length() (rooted_tree method), 35
track_locus() (multi_locus_genealogy method), 34
track_locus_genealogy() (haploid_highd method), 26

trait (clone attribute), 12
trait_weights (haploid_highd attribute), 21
treatment (hivpopulation attribute), 28
tree_edge (class in FFPopSim), 37
tree_key (class in FFPopSim), 36
tree_node (class in FFPopSim), 37
tree_step (class in FFPopSim), 36

weight_distribution (tree_node attribute), 37
write_genotypes() (hivpopulation method), 28
write_genotypes_compressed() (hivpopulation method), 28, 34